Calculating Optimal Jungling Routes in DOTA2 Using Neural Networks and Genetic Algorithms

Thomas E. Batsford
University of Derby
Derby, England
contacttomhere@hotmail.com

Abstract - In this paper the use of a Neural Network trained using Learning Algorithms is proposed to tackle one of the harder activities within a Multiplayer Online Battle Arena (MOBA) game; Jungling. The aim of the implementation was to find an optimal route around the jungle that can then be used by both players and bot AI whilst fighting in Dota2. To do this an accurate representation of the Dota2 Jungle was implemented and a Feed Forward Sigmoidal Neural Network used, in combination with various Genetic Algorithm techniques, for all decision making. Results show that some convergence toward an optimal route has been reached, however more work needs to go into the simulator itself before an accurate representation of a professional player can be achieved.

Keywords - MOBA, Dota, Dota2, Bots, Neural Network, Learning, Genetic Algorithms

I. INTRODUCTION

In 2003 a mod, arguably one of the most popular mods of all time, was released for Warcraft Three: Reign of Chaos [1] called Defence of the Ancients (DotA) by a man known only by his pseudonym 'Eul' [2]. He could never have imagined that today various different versions of DotA are played by millions of people all over the world. This, now huge genre, was first created as a mod for Starcraft [3] called Aeon of Strife. DotA was based on this Starcraft mod and began gaining a large following of players. In 2003 Blizzard released Warcraft Three: The Frozen Throne [4] which included a new and improved world editor, this allowed more people to experiment with the genre adding more features to the game and in the end lead to the creation of one of the most successful versions; DotA Allstars, by Steve "Guinsoo" Feak [2]. DotA began to get a huge following, and became one of the most played games on Battle.Net [5]. Today DotA can be played in a number of versions. The original DotA is still playable in Warcraft Three, but the majority of players have migrated to the three big names within the genre; League of Legends (LoL) [6], Heroes of Newerth (HoN) [7] and Dota2 [8], which is still in Beta.

In January 2013 S2 Games released HoN patch 3.0 which introduced bots to the game. These bots can be designed and implemented by modders using the LUA coding language. When a bot has been created, the developer can submit it to the HoN forums for approval and potential implementation into the game. The bots are currently used for people who are new to HoN to practice against, however S2 have said that they intend to create bot tournaments. "*McDaniel says, "We really think this will attract a unique community. We'd like to eventually create bot vs bot matches so modders can compete with each other trying create the best AIs."*" [9]. After some experience implementing bots into HoN it is clear that more complicated meta game activities can't be implemented with rudimentary Artificial Intelligence (AI) methods. This is when the idea of a Multiplayer Online Battle Arena (MOBA) bot controlled by a Neural Network and Genetic Algorithms was first realised.

II. RATIONALE

During the Game Behaviour lectures we were given an introduction to Neural Networks (NNs) and Genetic Algorithms (GAs) and both intrigued me greatly. The ability to achieve a result given any current world state seemed like a perfect fit for a MOBA bot, and the promise of GAs as a learning tool for NNs (along with the demonstration of Box Car 2D) made me want to implement my own versions.

As an avid MOBA player and programmer I was excited when the news of HoN bots was released, but slightly disappointed when I realised the scope for implementation was somewhat limited. Creative freedom was available but it was not possible to change too much of the underlying code and have your bot implemented into the game. The introduction of bots into HoN is in my opinion only the beginning of modding for MOBA titles. Depending on the success of HoN bots and the length of time before bot vs bot tournaments are introduced, I believe that other MOBA titles (Dota2 and LoL) will add something similar in the near future. I wanted to create a tool that would help make more advanced bot AI for the MOBA bot's of the future.

One of my preferred roles within a MOBA game is jungling / woodsing so it made sense to aim my NN and GA implementations at this area. Jungling in a MOBA game is easy to do, but hard to master. There are a number of advanced techniques that a player can employ to level up faster and help their team out, including pulling of creeps (non player controlled (NPC) units within the forested area of the game map) and stacking of creeps. Jungling within a MOBA game is explained in the first section of this paper followed by related work in the field of NNs and GAs and concluded with the results of the implementation and intended future work.

III. JUNGLING IN DOTA2

Jungling refers to the killing of neutral NPC creeps for gaining experience and gold. Jungling is usually done during the laning phase of the game (typically within the first 10 to 15 minutes) and allows both the jungling hero and solo lane hero to gain more experience and gold than their opponents. Figure 1 shows the areas that jungling takes place along with the solo lane locations that the jungler helps out on the Dota2 map. There are five types of neutral creeps in Dota2; easy, medium, hard, elder and a boss creep; Roshan. At the beginning of the game a jungling hero will only typically be able to kill the easy and medium creeps. Roshan requires a full 5 man team to kill and provides the most experience and gold out of any of the neutral NPC's [10].

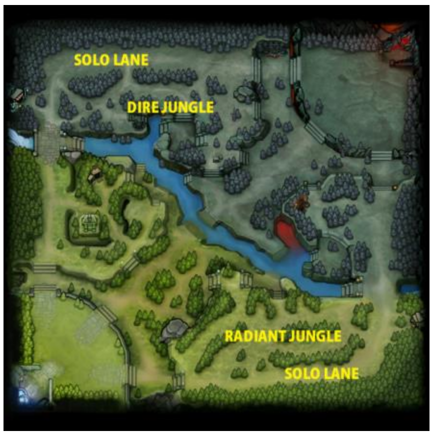


Fig. 1. Dota2 Jungle and Solo Lane Locations

After a creep camp (collection of NPC's that reside in a section of the jungle) dies, it respawns depending on a number of rules. First, the in-game clock must reach the minute mark, i.e. 1 minute, 2 minutes, 3 minutes etc. Second, the corpses of the previously killed NPC's at that creep camp location can no longer be there. Corpses take 24 seconds to despawn after the unit is killed. Finally no other unit can be in, or close to, the creep camp; this includes players, player controlled minions and some items that can be placed on the ground (an example being a ward which gives vision to the surrounding area). The type of creep camp that respawns when these conditions are met is randomised to add some variation and difficulty to the jungling experience. If the hero is killed by the creeps while he fights the camp then he will die and respawn back at the pool (area where items can be bought and health can be restored) after a timer has counted down [10].

Typicaly a hero will work his way around the jungle gaining experience and gold for each camp he kills, if the hero is low on health after fighting a creep camp he can heal back at the pool or use items to regain health. There are also a number of advanced tactics that the player can use to gain experience and gold faster and make the jungling easier, these are; Pulling creeps [10] and Stacking creeps [11]. The implementation of these advanced tactics is outside the scope of this project.

IV. RELATED WORK

Neural Networks (NNs) and Genetic Algorihms (GA)s are two types of learning techniques, each with their own strengths and weaknesses. They have both been researched for many years now, but have generally followed separate paths. Within the last 25 years the combination of the two has come far, starting with Whitley's [12] unsuccessful implementation in 1988 [13]. There are a number of ways to implement a NN, the main type being the Feed Forward NN. There are two types of feed forward NNs; the Single-Layer Perceptron (where the input neurons link directly to the output neurons) and the Multi-Layer Perceptron (where any number of hidden neurons can be placed between the input and output neurons) [14]. Multi-Layer NNs are the focus of this paper.

A. Feed Forward Neural Networks

Feed Forward NNs are networks in which each neuron (a node that represents a biological neuron) is only connected to the neurons in the subsequent layer, no neuron is connected backwards meaning the flow of the data moves only from left to right (as can be seen in figure 2). On the left hand side of a feed forward NN is the input layer. An NN can contain any number of input neurons within the input layer. Each input is represented by a normalised value and can refer to any kind of data, be it a boolean, float, integer or anything else that can be represented between the scale of 0 and 1. On the far right hand side of an NN is the output layer. Like the input layer, the output layer can contain any number of neurons. The user of the network must decide how many outputs the NN needs, and what each output value represents. Between the input and output layers are the hidden layers. A feed forward NN can contain any number of hidden layers, and each hidden layer can contain any number of neurons, there are however some rules of thumb to follow when deciding on the hidden layers, but the rules differ depending on the researcher [15] [16] [13]. A visual representation of a feed forward NN can be seen in figure 2.

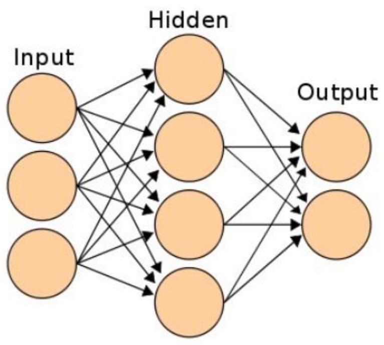


Fig. 2. Feed Forward Neural Network

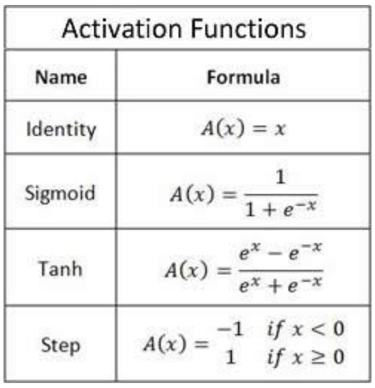
Each neuron within an NN (excluding the input layer) has an activation function. The output of a neuron is calculated by multiplying all of its incoming neurons by the weight along the connection and then summing the values, the resulting number is then passed through the activation function and the neuron fires, propagating its value to the next layer. Equation (2) shows how the output of a neuron is calculated [16].

$$Output = Activation\ function\ ((n1 * w1) + (n2 * w2) + (nn * wn)) \qquad \textbf{(2)}$$

Where:
n1 = output from the first incoming neuron
w1 = weight along the connection between the current neuron and the first incoming neuron
n2 = output from the second incoming neuron
w2 = weight along the connection between the current neuron and the second incoming neuron

etc.

There are a number of activation functions that are commonly used in feed forward NNs, these include; a linear function (identity), a sigmoidal function, a tanh function and a step function. Figure 3 shows how these functions are calculated. Heaton [16] justifies the use of each of these functions. He mentions that if a positive output from the NN is desired (i.e. a value between 0 and 1) it is best to apply a sigmoidal activation function. If a value that can be positive and negative is desired (between -1 and 1) then it is best to shift the sigmoidal function down on a graph by using a tanh function. Heaton also states that a linear function is the least used activation function and doesn't change the input in any way. There are two main types of training that are being researched in academia that can be applied to an NN; back propagation (a type of supervised learning [15]) and Genetic Algorithms (GAs) (a type of unsupervised learning). Back propagation is the method of training weights dependant on a number of predefined patterns that the user wants to replicate. The idea behind the method is that given a large range of potential inputs and the expected outputs, over time the weights in the NN can converge so that all patterns are completed successfully, i.e. the given inputs return the expected outputs. This means that when inputs are entered into the NN that are not part of a specified training set, an expected value is returned [16]. Training an NN using back propagation can take a long time; a much faster approach is the use of GAs.



Activation Functions	
Name	Formula
Identity	$A(x) = x$
Sigmoid	$A(x) = \frac{1}{1+e^{-x}}$
Tanh	$A(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
Step	$A(x) = \begin{matrix} -1 & if\ x < 0 \\ 1 & if\ x \geq 0 \end{matrix}$

Fig. 3. Activation Functions for NN Neurons

B. Genetic Algorithms

Genetic Algorithms (GAs) take their inspiration from evolution and the notion of "survival of the fittest". The core idea behind GAs is selecting and allowing the best candidates to breed with each other and create a new generation, this process continues until the best candidates begin to converge and thus the final solution is found and the network is trained. Montana and Davis [13] identify that there are 5 steps to using a GA; the same 5 steps can be seen in Koehn's [17] research and Sweetser's [18] research. A set of weights for the NN (from now on called a chromosome) is first initialised by randomisation. Once every weight has been randomised, the NN is run. After the simulation using the network is finished, a fitness value is assigned to the chromosome giving it a rating on how well it achieved the task. The fitness value is calculated with a fitness function. This is done a number of times to create what is called a population. The second step is selection. After a population is created, a number of chromosomes need to be selected from the population for breeding, these are called the parent chromosomes. There are a few methods for selection, and choosing the correct one is implementation specific. Below is a summary of each selection method [19] .

1) Roulette Selection

Roulette selection is the process of selecting random chromosomes as parents with a weighting dependant on their calculated fitness value. To do this, the population is first ranked by its fitness. Next, each chromosomes fitness value is summed together to get the total fitness of the population. A random number is then generated between 0 and the fitness value of the population. The population is looped over, summing the fitness value each loop. When the summed fitness value is greater than that of the randomised value the corresponding chromosome is selected as a parent.

2) Ranked selection

Ranked selection is similar to roulette selection, but is preferred when the chromosomes fitness values have large differences between them (for example, the top candidate may be taking up 90% of the population's fitness value). First the population is ranked in the same way as roulette selection, next each chromosome is given a value, starting at 1 for the lowest fitness chromosome then 2 for the second lowest, up to n for the best chromosome, where n is the number of chromosomes in the population. This method does allow lower fitness chromosomes to be selected but convergence can take longer due to the best chromosomes only having a slightly better chance of being selected than the worse ones.

3) Elitist Selection

Elitism refers to the selection of the best chromosome (or top few chromosomes) for use in the parent group. This is done because using other selection methods gives the best chromosomes a low chance of being selected. Once Elitist selection has been applied, the rest of the parents are selected using other methods.

Once the parents have been selected from the current population, breeding occurs. There are a number of ways that chromosomes can be bred, the most commonly used is crossover [20]. Crossover refers to the selection of a random point on each pairing of parents. Once the random point is chosen, child one is created by taking the fathers chromosomes up to the randomised point, and the mother's chromosomes after the randomised point. Child 2 is created in the same way, taking the first section from the mother and the second section from the father. This is called single point crossover. There are other methods of crossover including; multi point crossover (where a section between two points of the father chromosome is replaced with the same section from the mother, and vice versa) and uniform crossover (where random parts of the father chromosome are selected and then swapped with the corresponding part from the mother [19]).

Once two children have been created using crossover, another biological method is applied. Mutation is the process of changing weights in each chromosome given a probability. There are two main types of mutation. The first involves the adding of a random value to the weight, the random value is typically between the values used for the weight initialisation for the NN. The second is a full replacement of the weight with a value between the values used for the weight initialisation [18]. The rate at which mutation should occur is regularly discussed within academia, and is usually found by trial and error [21].

Once all of the parents have bred, and a new population of children has been created, the parents are re-added into the population and the process continues. The new weights are applied to the NN, the simulation is run and again each chromosome is evaluated against the fitness function. As this process continues, the chromosomes begin to converge towards the best fitness possible. When this happens, the weights have been successfully trained, and the NN is ready to be used in a full implementation.

V. IMPLEMENTATIONS

The Jungling Simulator, Neural Network (NN) and Genetic Algorithms (GA) have been implemented using XNA. Two simulation types have been implemented; a fast simulation for generating large populations for use in the GAs, and a watchable simulation for viewing the best fitness of each generation. A heads up display (HUD) has also been implemented to show the current health values and stats of the hero during the watchable simulation. The Dota2 jungle and statistics have all been calculated as accurately as possible. A Feed Forward Sigmoidal NN has been implemented to calculate the next position for the hero to move to. Various GAs have been implemented including single point crossover and mutation for generating new populations of weights to be used in the NN.

A. Dota2 and the Jungle

When attempting to find out the most optimal route around the jungle it is important to represent the game states as accurately as possible. This includes all Dota stats and battle simulations between the creeps. It was however, impractical to implement all possible features within the scope of this project. In order to arrive at a functional simulation the Dota2 and Jungle implementations needed to be reduced in scope, after all, implementing all features would result in a recreation of the entire game. A number of features have been stripped from the implementation in order to test the NN and GAs before a full implementation is explored. Removed features include; Items, Hero Abilities, Lane Creeps and advanced jungling techniques like pulling and stacking. The remainder of the implementations however were replicated as accurately as possible including the combat algorithm, hero / creep stats, experience values and movement around the map. The final implementation included; the movement of the hero around the jungle, waiting in locations around the jungle, fighting with creeps, healing in the pool and respawning when he Hero dies. The jungle implementations can be regarded as the base features that can easily be built on in the future.

Combat is calculated within a single frame. It is important to simulate the combat as accurately as possible due to time having the greatest impact on a jungler (due to creep respawning). One of the problems with calculating accurate fights is that every fight can be different, for example, when the hero takes on a creep camp, the creep that he starts attacking and thus kills first (baring in mind a creep camp can contain multiple types of creeps with varying damage, health and armour values) could be different each time. In order to model this accurately, weighted averages (weighted depending on each unit's health) of the creep camps stats were used. The equation (2) shows how the average of the weighted creep camps stats is calculated.

$$\textit{Average stat} = ((c1s * c1h) + (c2s * c2h) + \ldots (cns * cnh)) / th \qquad \textbf{(2)}$$

Where:
c1s = creep one stat
c1h = creep one health
c2s = creep two stat
c2h = creep two health
cns = creep n stat
cnh = creep n health
th = total health

This provides the average of each sat within a camp, given that any of the creeps can die in any order. In other words, if the fight was to be simulated multiple times, this value will be the result. This equation is applied to each stat that effects the battle simulation including armour, damage, attack speed and health regeneration. However, all creeps that are currently alive within the camp will be attacking the hero, and because creeps can die at any point during the fight, this means that a simple multiplication of *n creeps x average damage* was not suitable. Time needed to be taken into consideration as the state of the camp can change over time as creeps die. To calculate this, equation (3) was applied to the average damage from the first equation.

$$\textit{Average camp over the fight} = (ad * (1/2\, n(n+1))) / n \qquad \textbf{(3)}$$

Where:
ad = average creep damage calculated from the first equation
n = the number of creeps in the camp

The values from these two calculation functions were then tested against real in-game scenarios, and the results showed that these equations gave a very accurate representation of the creep camp's stats. The average times that the creep camps took to die was at the most 5 seconds out from the real game.

All of the Hero's stats within the simulation were replicated using the same algorithms as Dota2, this includes health, damage, mana, health regeneration, mana regeneration, attack speed and armour, which all depend on the hero's level and attributes; strength, agility and intelligence. Once all stat calculations were complete a battle simulation algorithm was calculated. The algorithm functions by calculating how much time it takes for the hero to kill the creep camp and then working out how much damage the camp would do in that time, if the damage is greater than the hero's current health then the hero dies.

The only calculated stat that wasn't used in the fight simulations was the hero's movement speed, but this did effect the simulation by reducing the time it took the Hero to move from one location to another. The movement paths in the simulation are pre-calculated, thus removing the time spent calculating A* paths when a simulation is run. In the fast simulation, the Hero is instantly moved to a location and the timer updated with how long it would have taken to get there. In the watchable simulation the only viewable aspect is the moving of the Hero along the paths (due to the fights and respawning being instant), this is the only part between the two types of simulations that differs, but it does allow the results of the NN and learning algorithms to be viewable.

To get as accurate as possible results, bonus stats are given to the hero each time it levels up, this represents item purchasing within the game. Because gold is not gained and items can't be bought in this implementation, it meant that at level 1 the hero would struggle to kill a lot of the creep camps. It also meant that despite the hero levelling up, harder creep camps could not be killed. To resolve this without changing the stats of the creep camps, artificial item buying has been represented by increasing the stat gains a hero receives when it levels up. Another component of the jungle that has been changed for the purpose of testing the GA and NN implementations is the randomised creeps that can spawn at each camp. For this implementation, the same creeps are spawned at the same camp every time, this means that given the same weights in the NN, the simulation will be replicated exactly.

B. Neural Network

A Feed Forward Sigmoidal NN has been implemented for calculating the best possible location to move to around the dota2 map given any current world state. In the input layer of the NN there are 16 neurons which correspond to; the current health of the hero, the current level of the hero, the amount of seconds through the current minute and then a neuron for each creep camps status – alive or dead. All of the input values are normalised before being passed into the NN. There is one hidden layers within the network containing 24 neurons. Varying amounts of neurons were tested however, and the findings of these tests are viewable in the results section. Because a number of rules of thumb exist and because "any rules of thumb are likely to fail in many cases" [15] it is hard to abide by only one rule. Champandard [15] says that a good place to start is with double the amount of input neurons, while Montana and Davis [13] use an increasing amount of neurons per layer, moving from the input layer containing 4 neurons to the first hidden layer containing 7 neurons and finally to second hidden layer containing 10 neurons. Blum mentions that "A rule of thumb is for the size of this [hidden] layer to be somewhere between the input layer size ... and the output layer size ...". Because "the chances of an efficient arithmetic solution appearing in the future are slim" [15], it is impossible to select the most optimal amount of neurons within a hidden layer based on other peoples research ad thus, trial and error must be used. The number of hidden layers chosen was based on Heaton's [16] research in the field, who states that "there can be any number of hidden layers" but that "for many practical problems, there is no reason to use any more than one hidden layer". Because both the number of neurons within a hidden layer and the number of hidden layers within an NN has not been universally agreed upon within academia, multiple combinations were tested and the results can be seen in the results section of this paper.

Each neuron within the NN is connected to every neuron in the previous layer. The activation function chosen to be used on each neuron was a sigmoid function. A sigmoid function was decided upon due to its output of a value between 0 and 1. Once the inputs have been fed into the NN, and an output is produced, the value between 0 and 1 is multiplied by the number of locations that the hero can move to. There is the pool and 13 creep camps on the map, so a value of between 0 and 14 is calculated. The value is then cast as an integer to arrive at a whole number value between 0 and 13 inclusive, if the value after multiplication is 14 it is rounded down to 13, this is due to the fact that any value less than 1 returns a result of 0 so any value greater than 13 should return a value of 13. The value is then fed into a switch case which outputs the location for the hero to move to next. If the hero moves to the pool, he is healed to full health and a new decision is made, if the hero moves to a creep camp and the creeps are alive, the hero will fight the creeps and then make a new decision (whether it survives the fight or not). If the selected location is the location that the hero is currently at then the hero waits for 1 second before making another decision, this allows the hero to wait for creep camps to respawn.

A full simulation finishes when the hero either reaches level 11 or the timer goes beyond 60 minutes. When the simulation finishes, the weights of the current NN are added to a generation data table along with the fitness value for the current simulation. The fitness value for the simulation is calculated simply by equation (4):

$$\textit{Total Gained Experience / Time Taken} \tag{4}$$

The value of this can range anywhere between 0 (when no experience is gained over the whole time period) and the optimal fitness value. The top end of this fitness function is what this implementation is attempting to calculate but a value of 7 would be deemed successful. A value of 7 represents reaching level 11 in around 16 – 17 minutes. This however is unlikely to be reached as hero items and skills are not taken into consideration when killing the jungle creeps and do offer a huge boost in the real game. After a number of generations are completed GA's are applied to the population to create the next population for testing.

C. Genetic Algorithms

The first time a generation is created random weights are assigned between the values -20 and 20 on each neuron connection within the NN. The simulation is run using these weights and the fitness and weights are saved, each set of weights will be referred to as a chromosome. After the entire generation has been calculated GAs are applied to create the next generation of chromosomes. There are three steps to applying the GA; selection of the chromosomes for breeding, crossover of each pair of chromosomes to create children and finally mutation of the child chromosomes. After all of these steps have been applied, the new generation of chromosomes are fed one at a time into the NN and a new simulation run to calculate the fitness value associated with each chromosome. This is repeated a number of times.

There are two types of selection method that have been applied to each generation; Elitist selection and Roulette selection. Different amounts of chromosomes are selected with each method in an attempt to find the best combination. Further information on how the different selection methods performed can be seen in the results section of this paper. Each generation has a total of 465 chromosomes / simulations. From this 465, 15 are chosen with the various selection methods for breeding. Each pair within this 15 creates two children using single point crossover.

Once the children have been generated mutation is applied at a rate of 8%. This mutation rate is quite high compared to what some academics recommend, Champandard [15] for example recommends a mutation rate of 0.1% while Reeves [20] states that since 1964 1 / l (where l is the number of weights) is the most optimal mutation rate. The mutation rate of 8% in this implementation is based on Haupt's [21] research into the most optimal mutation rates and selected population sizes for artificial NNs. He concludes that "*The results of the numerical experiments presented in this paper suggest that the best mutation rate for GAs lies between 5 and 20%*" [21]. The mutation rates were varied during testing, and the results of different mutation rates can be seen in the results section of this paper. After the children have been created, the parents are re-added back into the population and the cycle continues until convergence occurs.

VI. Results

To calculate the most optimal path around the jungle, and to test which Neural Network (NN) and Genetic Algorithm (GA) methods produced the best results, a number of tests were undertaken. The nature of GAs and NNs make it hard to calculate where an error lies if the results achieved are less than expected. To combat this error, simulations were done using different values for the following:

- Number of hidden layers in the NN
- Number of neurons within a hidden layer
- Number of chromosomes selected with Elitist / Roulette
- Mutation rate of the chromosomes

There are of course more variables that can be changed (than the ones selected for the tests) related to the NN and GAs but they are out of the scope of this project. These aforementioned variables for the tests were fixed for each simulation and included; the number of generations before the value was recorded - 20, the amount of simulations per generation - 465, the number of selected chromosomes for breeding – 15, the crossover method – single point, and the selection methods of the chromosomes – elitist (3) and roulette (12). For each test, the default values used were; one hidden layer, 24 neurons in the hidden layer, two elitist chromosomes and 13 roulette selected chromosomes and a mutation rate of 8%. The graphs for the various simulations can be seen below. Each test was run three times and an average taken, this accounted for any lucky random weights that could occur from the initial weight generation step. The best result from each experiment was then used to calculate the final result for the most optimal path. Each experiment contains 5 tests, each one of these tests is run 3 times to find an average for the test. Each time a test is run, 20 generations are created to get the best fitness value, and each generation consists of 465 simulations, this means that a total of 139500 jungling simulations are run per experiment.

A. One Hidden Layer – Increasing amount of neurons

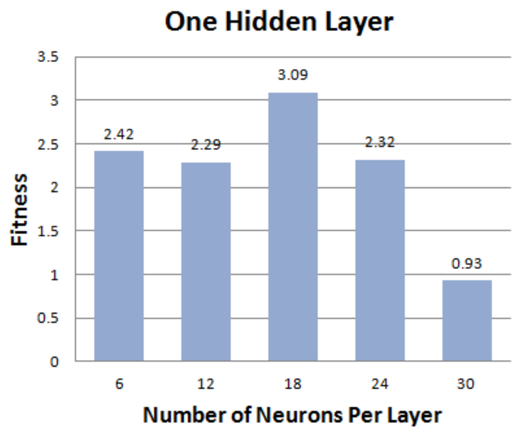


Fig. 4. Graph Showing Fitness Values Calculated Using One Hidden Layer

The first test undertaken was to determine which amount of neurons within a single layer in the NN returned the best fitness value. The graph above shows that when one hidden layer is used with 18 neurons within it, the best fitness was achieved.

B. Two Hidden Layers – Increasing amount of neurons

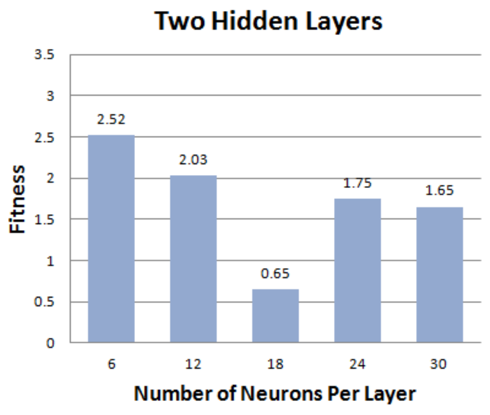


Fig. 5Graph Showing Fitness Values Calculated Using Two Hidden Layers

The second test used two hidden layers within the NN. The same amount of neurons were used in each layer. The amount of neurons in each layer was the same as in the first test. The results show that with two hidden layers, 6 neurons within each layer gives the best results, but the value of 2.52 is not better than the 3.09 achieved in the first test.

C. *Two Hidden Layers – More second layer neurons than first layer*

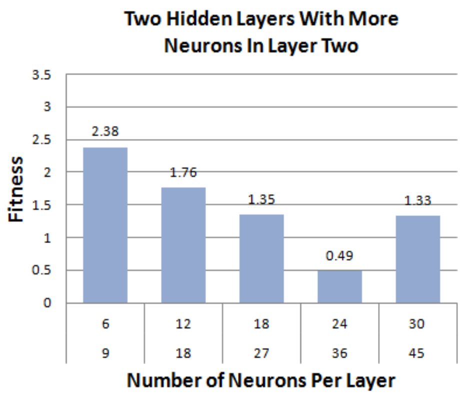


Fig. 6. Graph Showing Fitness Values Calculated Using Two Hidden Layers with Increasing Neuron Amounts In Layer Two

The final test that adjusted the layers within the NN followed the method used by Montana and Davis [13], increasing the amount of neurons within each subsequent layer. The first hidden layer within the NN followed the same pattern as within the first and second tests. The second hidden layer within the NN had 1.5 times the amount of neurons as the first hidden layer. The results show that the combination of 6 neurons and then 9 neurons provides the best fitness value. This is comparable to the second test where the 6 6 combination provided the best fitness value.

After the first three tests it is clear that the best combination of hidden layers and nodes within each layer is; one hidden layer with 18 nodes within it. The fitness value of 3.09 was the highest achieved. When the final simulations were run to calculate the optimal jungling route the one hidden layer with 18 neurons approach was be used.

D. *Increasing Mutation Rates*

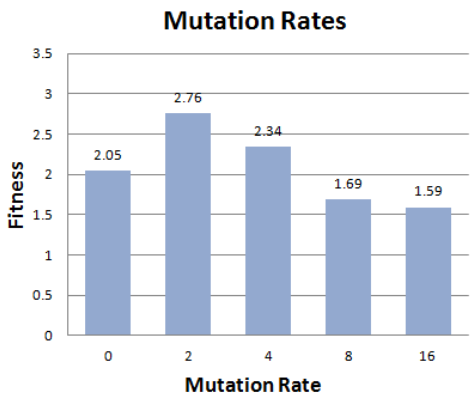


Fig. 7. Graph Showing Fitness Values Calculated Using Varying Mutation Rates

The fourth test was to calculate which mutation rate provided the best fitness. After averaging three results per test the best fitness value recorded was 2.76, this was using a mutation rate of 2%. This value was used in the final test when calculating the most optimal route around the Dota2 jungle.

E. *Increasing Amount of Chromosomes Selected With Elitism*

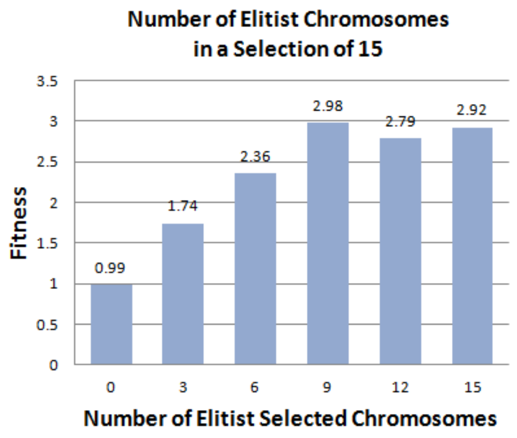


Fig. 8. Graph Showing Fitness Values Calculated Using Varying Amounts of Elitist Selected Chromosomes

The final test was done to calculate how many chromosomes, out of the 15 selected for breeding, should be selected using Elitism. The graph shows that as the amount of chromosomes selected with Elitism increases the fitness value increases. Despite this test showing that the best amount of Elitist chromosomes is 9 the nature of elitist selection can be a counter argument to this result. When a greater amount of elitist chromosomes are used in the parent group convergence happens at a faster rate, the problem is however that a local optimum can be hit. This means that the best potential fitness can't be reached. When calculating the final optimal route 9 chromosomes were selected with Elitism and 6 were selected with the Roulette method.

Once all the most optimal value for each tested variable was found the final tests were completed to work out the most optimal route around the jungle. Three tests were run using a single hidden layer with 18 neurons, a mutation rate of 2% and 9 chromosomes selected with Elitism and 6 with Roulette. The best fitness achieved was **4.43.** A fitness of 4.43 represents the hero reaching level 11 (the maximum reachable level in this implementation) in 24 and a half minutes. This value being reached means that the implementation of the NN trained by GAs was a success. Although the value isn't close to the 7 expected, it could be due to a number of reasons. The NN and GA variables have not been thoroughly tested, so some error may lie there. Another issue may be with the jungle and Dota2 implementations. Because the hero doesn't use skills or buy items it is impossible to compare this implementation with the real game. Another issue could be that the nature of NNs and GAs is based on random numbers. The initial weights are all randomised, along with the selected chromosomes for roulette and mutations. The final factor affecting the results is the variables that weren't tested including the generation sizes and crossover methodology. Despite this, a fitness value of 4.43 is impressive and shows that we are on the right lines to reaching an accurate optimal route.

VII. Conclusion

In this project an attempt to find the mot optimal route around the Dota2 jungle using a Neural Network (NN) trained with Genetic Algorithms (GAs) was made. A feed forward sigmoidal NN was implemented with three layers; an input layer with 16 neurons, a hidden layer with 18 neurons and an output layer with 1 neuron. The genetic algorithm methods used were; Elitist and Roulette selection, single point crossover to create child chromosomes and a mutation rate of 2% on each child. An accurate simulation of movement around the jungle and combat with creep camps has been implemented along with accurate stat calculations for the hero. Two types of simulation have been implemented, a fast simulation which can run through a single 60 minute Dota2 game in under a second and a Watchable simulation which allows the best results from each calculated generation to be viewable.

Five tests have been undertaken to calculate the best values for a number of NN and GA variables. Using the results from these tests the jungle simulation was run a number of times to arrive at the best fitness value and jungle route. The best fitness value arrived at was 4.43. The implementations of both the NN and GAs have been a success and convergence towards an optimal path can be seen. There are however a number of improvements that can be made. With a little more work, and possibly some more research in the field, hopefully a NN trained with GAs can be implemented into a real MOBA game bot to be used in bot vs bot tournaments.

VIII. Future Work

It is clear from the results and methodologies that there is still a large amount of work that could go into the project. To start with, a full implementation of the Dota2 jungle is needed. This includes all advanced jungling tactics as well as randomisation of jungle creeps. Also, hero abilities and items need to be implemented to give a more accurate representation of the most optimal route. To do this, current gold needs to be added as an input into the Neural Network (NN) hopefully allowing the Hero to learn when to go back to the pool to buy items. This implementation was also limited to only one hero – Axe (a Dota2 hero). One of the next steps, once a hero ability system is implemented, would be to add more heroes to the simulator.

More testing also needs to be done with the NN and Genetic Algorithms (GAs). There were a number of changeable variables that were not altered during the result gathering stage. Once these are tested and the perfect configuration of NN and GA implementations are finalised, the optimal path for jungling can be found.

The overriding goal of this project was to, at some point, use the calculated NN as an implementation in a HoN bot or Dota2 bot (when released). At this stage the calculated NNs are not accurate enough for a full implementation although I do believe that with a bit more work this goal can be reached. One large stretch goal would be to implement a complete bot using NNs and learning algorithms. Kolwankar [22] has begun looking into the use of Genetic Algorithms for MOBA bots, so hopefully more research in the field will make a full implementation possible in the not too distant future.

References

[1] Blizzard Entertainment, *Warcraft Three: Reign of Chaos*, Blizzard Entertainment, 2002.

[2] S. Feak and S. Mescon, "Postmortem: Defense of the Ancients," 2009. [Online]. Available: http://web.archive.org/web/20101207055840/http://www.gamasutra.com/view/feature/3966/postmortem_defense_of_the_ancients.php. [Accessed April 2013].

[3] Blizzard Entertainment, *Starcraft*, Blizzard Entertainment, 1998.

[4] B. Entertainment, *Warcraft Tree: The Frozen Throne*, Blizzard Entertainment, 2003.

[5] Blizzard Entertainment, *Battle.Net*, 1996.

[6] Riot Games, *League of Legends*, Riot Games, 2009.

[7] S2 Games, *Heroes of Newerth*, S2 Games, 2010.

[8] Valve Corporation, *Dota2*, Valve Corporation, TBA.

[9] G. Townsley, "A newbie in Heroes of Newerth: How patch 3.0 improves the MOBA experience," 2013. [Online]. Available: http://massively.joystiq.com/2013/01/21/a-newbie-in-heroes-of-newerth-how-patch-3-0-improves-the-moba-e/. [Accessed April 2013].

[10] Various Authors, "Jungling," 2011. [Online]. Available: http://www.dota2wiki.com/wiki/Jungling. [Accessed April 2013].

[11] Various Authors, "Creep Stacking," 2011. [Online]. Available: http://www.dota2wiki.com/wiki/Creep_Stacking. [Accessed April 2013].

[12] D. Whitley, "Applying Genetic Algorithms to Neural Network Problems," 1988.

[13] D. J. Montana and L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," Cambridge, 1989.

[14] P. Sweetser, "How to Build Neural Networks for Games," in *AI Game Programming Wisdom 2*, Charles River Media, 2004, pp. 615-625.

[15] A. J. Champandard, "The Dark Art of Neural Networks," in *AI Game Programming Wisdom*, Charles River Media, 2002, pp. 640-651.

[16] J. Heaton, Introduction to Neural Networks with C#, Heaton Research, Inc, 2008.

[17] P. Koehn, "Combining Genetic Algorithms and Neural Networks: The Encoding Problem," 1994.

[18] P. Sweetser, "How to Build Evolutionary Algorithms for Games," in *AI Game Programming Wisdom 2*, Charles River Media, 2004, pp. 627-637.

[19] M. Obitko, "IX. Selection (Introduction to Genetic Algorithms)," 1998. [Online]. Available: http://www.obitko.com/tutorials/genetic-algorithms/selection.php. [Accessed April 2013].

[20] C. Reeves, "Genetic Algorithms," in *Handbook in Metaheuristics*, Kluwer Academic Publishers, 2003, pp. 55-82.

[21] R. L. Haupt, "Optimum Population Size and Mutation Rate for a Simple Real Genetic," 200.

[22] K. S. V, "Evolutionary Artificial Intelligence for MOBA / Action-RTS," 2012.